Homework #7: Minimum Spanning Trees
CIS 121—Fall 2017

Due: Wednesday, November 15th, 2017

1 Conceptual questions about your implementation
10 points

Please answer the following questions (all running times should have a theta bound):

1. What is the running time of your implementation of Kruskal’s algorithm? Was there any part of the code that acted as a bottleneck? If so, how could that be avoided?

2. What is the running time of your implementation of Widest Path? Can you think of any alternative ways to solve the Widest Path problem?

3. What were some of the advantages and disadvantages of the Graph class using ints to represent vertices? Consider this question in the context of Client.java.

4. Where possible, we used a functional structure in this homework, encouraging you to avoid statefulness. For example, the BFS.getShortestPath() method is static and takes in the graph as an argument, ‘a la OCaml. Discuss some pros and cons of this kind of structuring.

Solution  Follows from homework implementation

Suggested rubric:

1. 3 points
2. 3 points
3. 2 points
4. 3 points

2 Connecting Two Components
10 points

True or False: Given any two components that are generated as Kruskal’s algorithm is running (but before it has completed), the smallest edge connecting those two components is part of the MST. Explain your answer.

Solution
False.

It could be the case that there is a shorter path between the two components that does not involve the single smallest edge connecting them directly. Here is a counterexample.
Assume the length of the edges is proportional to their weight. The black edges have been selected as part of the MST by Kruskal’s algorithm so far. Here, (2, 5) is the smallest edge directly connecting components \{1, 2\} and \{4, 5\}. But the path (1, 3) → (3, 4) has total length shorter than edge (2, 5).

**Suggested rubric:** 4 points for correct False answer; 6 points for correct counterexample or explanation

## 3 Cycle Property

**10 points**

Prove the following, known as the cycle property: Given any cycle in an edge-weighted graph (all edge weights distinct), the edge of maximum weight in the cycle does not belong to the MST of the graph.

**Solution** This is a proof by contradiction.

Given a graph \(G\) with cycle \(C\) and MST \(T\), suppose the edge \(e \in C\) with maximum weight does belong to \(T\), i.e. \(e \in T\).

If we remove \(e\) from the MST \(T\), then \(T\) becomes disconnected. Let \(S\) be the set of nodes to one side of the cut.

Since \(e\) was part of a cycle \(C\), then there must be some other edge \(f \in C\) having one edge in \(S\) and one edge outside \(S\). Thus adding \(f\) to the MST will re-connect the graph; the resulting tree \(T' = T \cup f - e\) is also a spanning tree.

But we know that since \(e\) had maximum weight \(w_e\) of all edges in \(C\), \(w_f < w_e\). Thus the sum of the weights in \(T'\) is less than the sum of weights in \(T\). This is a contradiction, since we started under the assumption that \(T\) was the MST for \(G\).

**Suggested rubric:** 10 points for a correct proof. Subtract points for any logical inconsistencies.

## 4 Two-edge Connectivity

**10 points**

A bridge in a graph is an edge that, if removed, would increase the number of connected components. A graph that has no bridges is said to be two-edge connected. Develop a linear-time DFS-based algorithm for determining finding all bridges in a graph (and determining whether it is two-edge connected).

**Solution** Recall that if we run DFS over a graph \(G\), some of the edges in \(G\) can be classified as tree edges (i.e. edges along the path of DFS traversal). Others can be classified as back edges, which point from any given node to one of its ancestors in the DFS tree. We can use these back edges to determine whether an edge is a bridge, under the key observation that if edge \((u, v)\) is a bridge, then there will be no back edges starting from the DFS subtree rooted at \(v\) that terminate at any of \(u\)'s ancestors (including itself). The existence of such a back edge would imply that there is another path connecting the subtree rooted at \(v\) to the component containing \(u\), and therefore cutting \((u, v)\) does not disconnect the graph. See Figure 4.
In order to build an algorithm that tests for back edges, we can run DFS while maintaining an array $\text{Low}[\cdot]$ with an entry corresponding to each node. The value pertaining to node $u$, $\text{Low}[u]$, intuitively is the node closest to the DFS tree root that you can get by taking any back edge from $u$ or any of its descendants. More formally, we define $\text{Low}[u]$ as follows:

$$\text{Low}[u] = \min(d[u], \min \{d[w] : \exists \text{ back edge } (v, w) \text{ where } v \text{ is a descendant of } u\})$$

Here, $d[u]$ gives the preorder numbering of node $u$ in the DFS tree.

To compute $\text{Low}[u]$ while running DFS, we first initialize $\text{Low}[u] = d[u]$ the first time we visit node $u$. Then, for each undiscovered neighbor $v$ of $u$ (indicating $(u, v)$ is a DFS tree edge), we run DFS from $v$ and set $\text{Low}[u] = \min(\text{Low}[u], \text{Low}[v])$ when DFS from $v$ completes. For each already discovered neighbor $w$ of $u$ (indicating $(u, w)$ is a back edge), we set $\text{Low}[u] = \min(\text{Low}[u], d[w])$.

Finally, once DFS completes for the start node, we can check for the bridges using the $d[\cdot]$ and $\text{Low}[\cdot]$ arrays in linear time. For each edge $(u, v)$, we check whether $d[v] = \text{Low}[v]$. If true, then $(u, v)$ is a bridge because there is no back edge from any descendant of $v$ to $u$ or any of its ancestors.

The runtime of this algorithm is $O(V + E)$, the same as for DFS.

Pseudocode (not required for students but provided for clarity) is below.

**Suggested rubric:** 10 points for a correct $O(E + V)$ algorithm.

5 points if they give an $O(E(E+V))$ algorithm (e.g. one edge at a time, delete the edge and run DFS).

-3 if they do not use DFS.

Half points for recognizing that existence of a back edge from the subtree rooted at $v$ to an ancestor of $u$ (or $u$ itself) means $(u, v)$ is not a bridge, without providing the full algorithm.
DFSvisit(u) {
    discovered[u] = True;
    Low[u] = d[u] = ++time; // set discovery time and initialize Low
    for each (v in Adj(u)) {
        if (discovered[v] == False) { // (u,v) is a tree edge
            pred[v] = u; // v's parent is u
            DFSvisit(v);
            Low[u] = min(Low[u], Low[v]); // update Low[u]
        } else if (v != pred[u]) { // (u,v) is a back edge
            Low[u] = min(Low[u], d[v]); // update Low[u]
        }
    }
}

findAllBridges(G) {
    time = 0;
    for each (u in V) // init
        discovered[u] = False;
    for each (u in V)
        if (discovered[u] == False) // undiscovered vertex?
            DFSvisit(u); // ...start new search here
    foundBridge = False;
    for each (v in V) { // check for the bridges
        u = pred[v];
        if (u != null and d[v] == Low[v])
            foundBridge = True;
            output (u,v) as a bridge
    }
    if (foundBridge == False)
        output that there are no bridges
}

5 Critical Edges
10 points

An MST edge whose deletion from the graph would cause the MST weight to increase is called a critical edge. Show how to find all critical edges in a graph in time proportional to E log E. Note: This question assumes that edge weights are not necessarily distinct (otherwise all edges in the MST are critical).

Solution
The key observation here is that an edge $e$ in graph $G$ is critical if and only if it is a bridge in the subgraph of $G$ containing all edges with weight less than or equal to the weight of $e$.

A proof of the statement above:

- Forward direction: If an edge $e$ is a bridge in the subgraph of $G$ containing edges with weight less than or equal to the weight of $e$, then it is critical.

  Suppose not, i.e. an edge $e$ is a bridge in the subgraph but not critical. By definition of critical, this means that there is another edge $f$ in the subgraph with weight equal to $e$ that could replace $e$ in the
MST. But because we’ve assumed \( e \) is a bridge, there cannot exist such an edge \( f \) as that would make
the subgraph two-edge connected. A contradiction.

- Backward direction: If an edge \( e \) is critical in graph \( G \), then it is a bridge in the subgraph of \( G \)
containing edges with weight less than or equal to that of \( e \).

This follows from the definitions of critical and bridge. If \( e \) is critical, then there are no edges \( f \) with
weight less than or equal to \( e \) that cross the cut of the MST created by removing \( e \). Since our subgraph
contains exactly that subset of edges, \( e \) is also a bridge of that subgraph by definition.

Now that we have established that a critical edge \( e \) is a bridge in the subgraph of edges with weight equal
to or less than that of \( e \), we can search for critical nodes in \( G \) using a two step process. First, we run
Kruskal’s algorithm over \( G \) to find the MST. Next we run DFS over the MST (accomplished by ordering
each node’s adjacency list to put MST edges first) while searching for bridges using the answer to Q4 above.
The implementation of \texttt{DFSvisit} requires one slight modification. Instead of keeping track of all back edges
to calculate \( \text{Low}[u] \), we only consider back edges if they have weight \( w \) equal or less than that of \((u, v)\):

\[
\text{Low}[u] = \min(d[u], \min\{d[x] : \exists \text{ back edge } (v, x) \text{ where } v \text{ is a descendant of } u \text{ AND } w_{v,x} \leq w_{u,v} \})
\]

Practically, we can update our Q4 algorithm above to accommodate this change by maintaining an additional
node-indexed array, \( \text{LowWgt}[\] \( \), which for node \( u \) stores the minimum weight of an edge pointing from a
descendant of \( u \) to \( \text{Low}[u] \). We initialize \( \text{LowWgt}[u] \) as \( w_{t,u} \) where \( t \) is the parent of \( u \) in the DFS tree.
Then, when setting \( \text{Low}[u] \) in \texttt{DFSvisit}(\( u \)) for newly discovered nodes \( v \), we only update \( \text{Low}[u] = \text{Low}[v] \)
if \( \text{Low}[v] < \text{Low}[u] \) and \( \text{LowWgt}[v] \leq \text{LowWgt}[u] \). When updating \( \text{Low}[u] \) for back edges \((u, v)\), we only set
\( \text{Low}[u] = d[v] \) if \( d[v] < \text{Low}[u] \) and \( w_{u,v} \) is less than or equal to \( \text{LowWgt}[u] \).

\textbf{Suggested Rubric} Give partial credit (6 points) for a solution that runs in time greater than \( O(E\log E) \)
(e.g. run Kruskal’s, and if adding an edge \((u, v)\) introduces a cycle then check to see if there are other edges
along the current MST path from \( u \) to \( v \) with equal weight.